

**C PROGRAMMING STANDARDS AND GUIDELINES
VERSION U (UNIX & OFFSPRING)**

Thomas Plum

PLUM HALL

RECEIVED

SEP 23 1983

I. M. A.

C PROGRAMMING STANDARDS AND GUIDELINES

VERSION U (UNIX AND OFFSPRING)

Thomas Plum

Plum Hall Inc
1 Spruce Avenue
Cardiff, NJ 08232
609-927-3770

CONTENTS

Preface

Chapter 0

Introduction

0.1standards

Standards and guidelines

Chapter 1

Data and Variables

1.1lexdata

Lexical rules for variables

1.2names

Choosing variable names

1.3stdtypes

Standard defined-types

1.4constants

Maintainability of constants

1.5wordsize

Word and byte size

1.6byteorder

Byte ordering

1.7charconsts

Character constants

1.8ptrtypes

Pointer types

1.9ptrconv

Pointer conversions

Chapter 2

Operators

2.1lexops

Lexical rules for operators

2.2evalorder

Allowable dependencies on evaluation order

2.3bitparen

Bitwise operators and parentheses

2.4rightshift

Right shift and unsigned data

2.5sideorder

Order of side effects

Chapter 3

Control Statements

3.1lexctl

Lexical rules for control structure

3.2while

"while" and the N+1/4 - time loop

3.3loopinvar

Designing with loop invariants

3.4elseif

Multiple-choice constructs

3.5control

Restrictions on control structures

3.6structure

Matching program structure to problem structure

Chapter 4

Functions and other modules

4.1lexfns

Lexical rules for functions

4.2headers

Project-wide standard header files

4.3files

Size for source files

4.4includes

Includes at head of file

4.5stdflags

Standard compile-time flags

4.6nonest

No nested header files

4.7noinit

No initializations in header files

4.8coupling

Methods of coupling modules together

4.9cohesion

Cohesion and meaningful functions

4.10libfns

File structure for library functions

4.11portlib

Use of portable library

4.12environ

Non-portable environment features

4.13fnsiz

Suggested size of functions

4.14macros

Writing macros

Chapter 5

General Standards

5.1compilers
5.2comments
5.3specs
5.4reviews
5.5defensive

Avoiding non-portable compiler features
Suggested use of comments
Specifications
Code reviews
Defensive programming

Chapter 6

Appendix

6.1v7
6.2ws
6.3v6
6.4refs
6.5index

Non-standard features of UNIX V7 and III compilers
Non-standard features of Whitesmiths compiler
Non-standard features of UNIX V6 compiler
References
Index

PREFACE

C is a highly portable language which generates efficient code for a wide variety of modern computers. It was originally implemented on the UNIX* operating system for the DEC PDP-11 by Dennis Ritchie. Within Bell Laboratories it is widely used for systems and application programming. In the years since it was made available to universities and commercial organizations, it has proved valuable for systems programming, switching and communications, microprocessors, text processing, process control, and test equipment. Its audience has been widened by the compilers from Whitesmiths, Ltd. that run on many operating systems besides UNIX. Whitesmiths' Idris* operating system (on which this book was composed) is, like UNIX, almost entirely written in C.

Programming standards can be valuable to any organization producing programs in C. Its compact notation and absence of restrictions can be used in an undisciplined fashion to produce programs unreadable to any but the original author -- if indeed the author can read them after the passage of time! A uniformity of style can make the thankless task of the maintainer much easier. The code can be modified much easier when standards are followed.

In addition to aiding consistency, standards also enhance portability of the source code, since one of the important virtues of the C language is its combination of portability and efficiency.

However, achieving portability requires attention to a small set of problem areas, which are addressed by the portability standards in this book.

Disagreements over programming style have been a primary obstacle to teams attempting to work closely together. One suggestion for preventing style arguments is for each project to choose its own standard in the early phases of the project. The layout of this book was chosen to facilitate its use in a pick-and-choose fashion. Each section is named (in the style of UNIX/Idris manuals) as well as numbered, for ease of later reference. The author asks in accordance with copyright laws that this book not be run through the copying machine; Plum Hall Inc will make available on a reasonable license arrangement both hard-copy and machine-readable originals for projects that wish to incorporate this material in their own standard.

In this Version U of the C Standards and Guidelines, the usage of types, function names, and indentation conforms to the format of the UNIX manuals published by Bell Laboratories. This book is also available in the format of the manuals from Whitesmiths,

*Trademarks: Idris of Whitesmiths, Ltd.; UNIX of Bell Laboratories.

Ltd., in Version W. The discussions of portability, however, apply to both systems.

For thoughtful comments and suggestions on earlier drafts of this material, I am indebted to Tom Bishop, Debbie Fullam, Joan Hall, Brian Kernighan, Bill Koenig, Mark Krieger, Eli Lamb, Ian MacLeod, Bill Masek, Paul Matthews, Bill Plauger, Ed Rathje, and Chaim Schaap.

Thomas Plum

Plum Hall Inc
1 Spruce Avenue
Cardiff, NJ 08232
609-927-3770

NAME

0.1standards - standards and guidelines

STANDARD

Criteria labeled as "STANDARD" are mandatory for all code included in a product.

The need for exceptions may occasionally arise, but the exception requires a specific justification, and the justification will be documented with the source code.

Project-wide exceptions to the standards may be justified and should be documented as an appendix to the standard.

Criteria labeled as "GUIDELINE" are recommended practices. Experience has shown that differing approaches can coexist in these areas. It is expected that, in general, a majority of programmers will follow the guidelines, so that they represent a widely-accepted pattern.

NAME

1.1lexdata - lexical rules for variables

STANDARD

Variable names should be written all in lower case. The language requires names to be distinct within 8 characters, but longer names can be useful for readability. (For portability, externals should be distinct within 6 characters.) All names should be explicitly declared. The sequence of declarations should be as follows:

external names, alphabetized by name within type;

other names, similarly alphabetized by name within type.

Initializers should be written using the equal-sign:

```
char s[] = "abc";
short c1 = MAX;
```

Initializers of structures, unions, and arrays should be formatted with one row per line:

```
static short x[2][5] =
{
    1, 2, 3, 4, 5,
    6, 7, 8, 9, 10,
};
```

Declarations should have only one space between type and variable name, and comments are attached with at least one tab:

```
bool mpflg = NO;    /* preprocessed macros flag */
bool ff;            /* fork flag */
```

JUSTIFICATION

The rules pinpoint the location of declarations, avoid conflicts of upper- and lower-case names, and encourage documentation of the meaning of variable names.

ALTERNATIVES

Code destined for compilation on "old" UNIX V6 compiler should have no equal-signs on initializers:

```
char s[] {"abc"};
short c1 MAX;
```

Variable names should be aligned in a tabbed column, and descriptive comments should be attached at a lined-up tab position:

```
bool      ff;                /* fork flag */
bool      mpflg = NO;        /* preprocessed macros flag */
```

NAME

1.2names - choosing variable names

GUIDELINE

Names should be chosen to be meaningful, and their meaning should be exact and should be preserved throughout the program.

For example, variables which count something should be initialized to the count which is valid at that point; i.e., if the count is initially zero, the variable should be initialized to 0, not to -1 or some other number.

This means that each variable has an invariant (i.e. unchanging) meaning -- a property that is true throughout the program. The readability of the code is enhanced by minimizing the "domains of exceptions", which are the regions of the program in which the invariant property fails.

Abbreviations for meaningful names should be chosen by a uniform scheme. For example, use the leading consonant of each word in a name.

Names should not be re-defined in inner blocks.

A special case of meaningful names is the use of standard short names like "c" for characters, "i", "j", "k" for indexes, "n" for counters, "p" or "q" for pointers, and "s" for strings.

In separate functions, variables with identical meanings can have the same name. But when the meanings are only similar or coincidental, use different names.

Names over four characters in length should differ by at least two characters:

```
systst, sysstst /* easily confused */
```

JUSTIFICATION

Readability of the code is greatly enhanced by the reader's ability to construct natural assertions about the meaning of names anywhere they appear in the code.

EXAMPLE

```
short nc; /* number of characters */

nc = 0;
while (getchar() != EOF)
    ++nc;
```

NAME

1.3stdtypes - standard defined-types

STANDARD

Programs should use a project-wide standard set of data-type names.

The set of standard types presented here are a mixture of standard C types (sometimes with usage restrictions) and defined-types defined by the header file <stdtyp.h> (presented later in this section).

float - float (single)

double - float (double)

long - a 32 bit (or larger) signed integer used for a quantity

lbits - a 32 bit (or larger) integer used for a bit manipulation

short - a 16 bit (or larger) signed integer used for a quantity

ushort - a 16 bit (or larger) unsigned integer used for a quantity

bits - a 16 bit (or larger) integer used for bit manipulation

char - an 8 bit (or larger) item used only for characters

tiny - an 8 bit (or larger) signed integer

utiny - an 8 bit (or larger) unsigned integer used for a quantity

tbits - an 8 bit (or larger) integer used for bit manipulation

tbool - an 8 bit (or larger) integer, but only tested against zero

metachar - a 16 bit (or larger) augmented character

void - a function that returns no value

bool - int, but only tested for zero or non-zero

unsigned - unsigned int, which can hold sizeof anything

FILE - a structure for buffered I/O calls (from <stdio.h>)

Even if other exceptions are allowed, the C data type int should not be used for data declarations, since it leads to non-portable programs whose capacity differs according to word size. The only appearance of int should be in declarations of function returned value type.

Integer-type function parameters should be typed by default; no declaration should appear for them. This is the convention followed by the V7 manuals; System III manuals later opted for explicit int declaration of parameters. Our choice is dictated by portability considerations described below.

Programs must use the semantically correct data-type name, even where several similar names map onto the same raw C language type.

All bitwise operations must be done on data that has one of the bits types; this alerts the reader to the bitwise nature

of the operations. See the section 2.4rightshift for the "long unsigned right shift" macro LURSHIFT.

Never redefine a defined-type.

Unfortunately for portability, not all compilers support the data types ushort, utiny, and tiny. For maximum portability, each rvalue use of variables declared with these types should use the corresponding macros USHORT, UTINY, and TINY. Each such macro is specific to the machine/compiler combination being used, and generates code only when the underlying type is not directly supported.

Although defined-types are presented here for signed and unsigned 8-bit numbers, programmers should avoid them except for the special cases where they are needed: very large arrays of small numbers, and code designed for portability to 8-bit microprocessors. Ordinary short integers are the preferred type for ordinary counters, and bits is the preferred type for bitwise data.

JUSTIFICATION

Careless use of int can lead to programs which work on one machine but not on another machine that has different word size.

Several bugs in C programs of an earlier generation were found by rigorously type-defining and consistency-checking such interfaces as function arguments, function returned values, and structure definitions. This semantic readability argument is the only justification for the defined-types lbits, bits, tbits, metachar, bool, and tbool, because the underlying types are not subject to any portability problems for their restricted usage.

The case is different for the types ushort, tiny, and utiny. Portability problems (both machine and compiler) are a main reason for these distinctions. Different versions of the compiler have introduced data types for unsigned char, unsigned short, and even (from Whitesmiths), unsigned long. And using the UNIX compilers, different machines may treat the type char as signed or unsigned. If programs are written with these raw C types, they must be edited by hand before porting to another compiler that does not support a new type. However, if a set of defined-types is consistently used, then the defined-type can be targeted to the new type when it exists, and be targeted to an older base type otherwise. For true portability, such defined-types must be augmented with rvalue macros:

USHORT(n) produces an unsigned value from 16 or more bits of n
TINY(n) produces a signed value from 8 or more bits of n
UTINY(n) produces an unsigned value from 8 bits (or more) of n

These defined-types are written in lower case to enhance

their compatibility with standard C types.

ALTERNATIVES

The prevailing alternative in the UNIX world is the use of standard C types. These are, however, prone to the portability and readability problems described above.

The common Whitesmiths usage, as described in Version W of these standards (Plum, 1982), is a set of upper-case names which completely supercede the standard C types. These names are, of course, inconsistent with the descriptions of C programs as found in UNIX-derived manuals.

EXAMPLE

```
bool main(argc, argv)
unsigned argc;
char *argv[];
{
    bool code;
    double x, y, z;
    ...
}
```

The following page presents the file <stdtyp.c>.

Header file <stdtyp.c>:

```
#ifndef FOREVER
/* The file is enclosed in a "ifndef FOREVER ... endif" wrapper */
/* stdtyp.h - standard defined-types
 * This file should either be customized to the intended
 * compiler/machine environment, or parameterized via -D flags
 * on the compile step:
 * -D USHORT if compiler supports unsigned short
 * -D TINY if char is signed
 * -D UTINY if compiler supports unsigned char
 * -D VOID if compiler supports void
 */
typedef char tbits, tbool;
typedef int bool;
typedef long lbits;
typedef short bits, metachar;
#ifdef USHORT
typedef unsigned short ushort;
#define USHORT(n) (n)
#else
typedef short ushort;
#define USHORT(n) ((unsigned)((n) & 0xFFFF))
#endif
#ifdef TINY
typedef char tiny;
#define TINY(n) (n)
#else
typedef char tiny;
#define TINY(n) (((n) & 0x80) ? (~0x7F | (n)) : (n))
#endif
#ifdef UTINY
typedef unsigned char utiny;
#define UTINY(n) (n)
#else
typedef char utiny;
#define UTINY(n) ((n) & 0xFF)
#endif
#ifdef VOID
typedef int void;
#endif
#define FOREVER for(;;)
#define YES 1
#define NO 0
#define LURSHIFT(n, b) (((long)(n) >> (b)) & (0x7FFFFFFF >> (b-1)))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define ABS(a) ((a) > 0 ? (a) : -(a))
#define FAIL 1
#define SUCCEED 0
#endif
```

NAME

1.4constants - maintainability of constants

STANDARD

Any constant which might change during revision or modification should be "manifest" ("clearly apparent to the sight or understanding; obvious"). Specifically, it should be given an upper-case name (via #define).

If it is only used in one file, it should be #defined at the head of that file; if used in multiple files, it should be #defined in a header-file.

Standard manifest constants should be used where appropriate. Examples from <stdio.h>:

BUFSIZ	the size of a standard file buffer
NULL	zero, as a pointer value
EOF	the end-of-file return value from getchar

JUSTIFICATION

Constants that are hard-coded (otherwise known as "magic numbers" because they mysteriously appear with no explanation) are hard to locate when modifying the program. Furthermore, instances of "constant minus one" or "constant plus one" are even more elusive to the maintenance programmer.

EXAMPLE

```
if (99 <= index)    /* wrong - no explanation, hard to modify */

#define LIMIT 99
...
if (LIMIT <= index) /* right */
```

NAME

1.5wordlen - word and byte size

STANDARD

A C program should assume the following sizes for data:

char 8 bits (or more)

short 16 bits (or more)

long 32 bits (or more)

A program should never rely on data size (or typecasts) to truncate expressions to a specific number of bits.

Bitwise constants can be made more portable by relying on the "bitwise negation" operator to set high-order one-bits:

```
~0       /* all one bits */
```

```
0177777 /* only 16 one-bits */
```

JUSTIFICATION

The main object of this standard is portability. There are C compilers with chars of 8, 9, or 10 bits, with shorts of 16, 18, 20, and 36 bits, and with longs of 32, 36, and 40 bits.

Even if one's target machines all have the same word size, different compilers handle the size of rvalues differently. For example, the BTL V7 compiler guarantees truncation of typecasts, register variables declared shorter than int, arguments declared shorter than int, and function returned values. Whitesmiths compiler does not guarantee such truncation.

NAME

1.6byteorder - byte ordering

STANDARD

Portability demands that programs not depend upon the order of bytes within an integral or floating number.

For example, the address-of operator should not be typecast into a pointer of different size:

```
char *p;  
short n;
```

```
p = (char *) &n;    /* gives machine-dependent byte */
```

Addresses passed as arguments to functions should agree in type with the declared formal parameters of the function. One exception is explicitly allowed: integer arguments to functions may be shorter than int size.

Taking the address of a function parameter must be done carefully, because C widens chars and shorts to int parameters. Whitesmiths compiler correspondingly widens parameter declarations to int (so that the stack layout is not dependent on byte-order). Any such parameter whose address is computed in the function should not be declared with any type smaller than int, for maximum portability to other compilers.

Binary data written on one machine will be portable to another machine only if byte-ordering dependencies are eliminated. A canonical ordering for binary data must be chosen. For historical reasons, the PDP-11 byte-ordering is the best choice for a canonical ordering. Before writing binary data, it should be converted to this canonical order; after reading canonical binary data, it should be converted back. The same considerations apply to floating data, but must be addressed locally.

JUSTIFICATION

It is possible to write C programs which will run on machines with different byte-orders, but one must be aware of the cases in which byte-order must be compensated for.

NAME

1.7charconsts - character constants

STANDARD

Portability requires that character constants not contain more than one character. The differences in machine byte-order may cause multi-character constants to differ either in numeric value or in character sequence.

EXAMPLE

```
short crlf = '\r\n';    /* bad - uses char constant */  
  
char crlf[] = "\r\n";   /* ok - uses string constant */
```

NAME

1.8ptrtypes - pointer types

STANDARD

Pointer entities (variables, function return-values, and constants) should be explicitly declared with pointer type.

JUSTIFICATION

Carelessness in pointer types (most notoriously, treating pointer as though it were int) can cause needless warnings from automated type-checkers such as "lint".

NAME

1.9ptrconv - pointer conversions

STANDARD

Programs should contain no pointer conversions, except for the following safe ones:

NULL may be assigned to any pointer.

Allocation functions (e.g. malloc) will guarantee safe alignment, so the returned value may be assigned to any pointer. Always use sizeof to specify the amount of storage to be allocated. And when allocating space for arrays of structures, remember that there may be holes between the structures.

Pointers to an object of a given size may be converted to a pointer to an object of smaller size and back again without change. For example, a pointer-to-long may be assigned to a pointer-to-char variable which is later assigned back to a pointer-to-long. Any use of the intermediate pointer, other than assigning it back to the original type, gives machine-dependent code.

JUSTIFICATION

Other conversions will be compiler-dependent or machine-dependent, or both.

EXAMPLE

```
short *pi;  
char *pc;
```

```
pi = NULL; /* always ok to assign NULL */
```

```
pi = malloc(sizeof short); /* always ok to assign malloc() */
```

```
pc = pi; /* always ok to assign to smaller-type pointer */  
pi = pc;
```

NAME

2.1lexops - lexical rules for operators

STANDARD

The unary operators should be written with no space between them and their operands. (Exception: sizeof, a keyword, is followed by a blank.)

The primary operators ">", ".", and "[]" should also have no space around them.

The assignment operators must always have space around them, and so must the conditional operator.

Commas (and semicolons) should have one space after them.

The other binary operators should generally be written with one space on either side of the operator. Occasionally, these operators may appear with no space around them, but the operators with no space around them must bind their operands tighter than the other operators around them.

The intention of the standard is to ensure that the common or habitual way to format binary operators is with surrounding space; exceptions in the interest of readability are recognized and allowed, but the exceptions should not be the majority of a function.

Keywords (if, while, for, switch, return, sizeof) should be followed by one blank.

Parentheses after function names should have no blank before them.

Arguments to functions are written with no space after the opening parenthesis and no space before the closing parenthesis.

JUSTIFICATION

Readability of the code is enhanced by a uniform layout of the operators.

Spaces are related to precedence by the following observation:

Psychologically, spaces connote LOOSER BINDING than the absence of spaces. Consider the difference in meaning between "light housekeeper" and "lighthouse keeper". The same principle labels this code as misleading:

```
n = a+b * c;    /* misleading spacing */
```

Automated processing of program text by editor programs and other text-searchers is possible only if blanks are rigorously formatted.

Also, if binary operators are routinely written with no space around them, there is a notorious ambiguity that is bound to happen sooner or later. The statement "x=-1;" will be taken as an instance of the assignment operator "=-", decrementing x by 1.

EXAMPLE

```
x = y * (z + 2);

for (i = 1; i < argc; ++i) /* K + R, pg 111, (mod.) */
    printf("%s%c", argv[i], (i < argc-1) ? ' ' : '\n');

if (x == 1)
    ...

void show(x)
    ...
    y = val(z);
```

NAME

2.2evalorder - allowable dependencies on evaluation order

STANDARD

Programs may not depend upon the order of evaluation of expressions, except as guaranteed by C for the following:

```
expr1, expr2
expr1 ? expr2 : expr2
expr1 && expr2
expr1 || expr2
```

All of these guarantee that `expr1` will be computed before `expr2`.

In addition, if we define a full expression to be an expression which is not a subexpression, C guarantees that evaluation of the full expression will be complete before any other expression is evaluated.

Commas when used to separate arguments in a function invocation or to separate names in a declaration, are not operators, and no evaluation order should be assumed. Only the operator comma will guarantee order of evaluation, as in this example:

```
tmp = x[i], x[i] = x[j], x[j] = tmp;
```

JUSTIFICATION

Code which depends on the order of evaluation may perform differently with different compilers or different machines, may perform differently when the code generator for a given compiler is changed at some future time, and may even perform differently with the same compiler if changes to the program affect the allocation of registers.

EXAMPLE

```
if (j < MAXBOUND && a[j] == TARGET)    /* ok order-dependence */
printf("%d:%d\n", ++i, ++i); /* bad order-dependence */
```

NAME

2.3bitparen - bitwise operators and parentheses

STANDARD

Bitwise operators (& | ~ ^ >> <<) should be explicitly parenthesized when combined with other dyadic operators.

JUSTIFICATION

Experience has shown their precedence to be misleading.

EXAMPLE

```
if ((status & MASK) != SET)    /* parentheses are needed */  
  
field = (((unsigned)w >> OFFSET) | FLAG); /* parentheses for readability */
```

NAME

2.4rightshift - right shift and unsigned data

STANDARD

Right-shift will not be applied to signed operands.

To insure uniform application of this standard, always use the macro LURSHIFT() when right-shifting long integers, and use the cast (unsigned) when right-shifting other integers.

JUSTIFICATION

Sign-extension is either compiler-dependent or machine-dependent. Also, right-shift as "pseudo-divide" gives wrong results for negative numbers on twos-complement machines.

Be careful when right-shifting the expression result of other operations. The expression result may be signed even though the operand(s) are not signed. For example, if variable t is an unsigned char, the expression ~t will be a (signed) int; thus

```
(~t) >> 4
```

will take place as a signed operation. The unsigned shift should be obtained by

```
((unsigned)~t) >> 4
```

EXAMPLE

```
-1 >> N      /* bad - gives -1 result for any N */
```

```
(unsigned)~0 >> N /* good - guarantees unsigned shift */
```

NAME

2.5sideorder - order of side effects

STANDARD

Programs must not depend upon the order in which side effects take place. In particular, the assignment and increment/decrement operators may alter the memory at unpredictable times during the evaluation of the expression.

JUSTIFICATION

Programs that depend upon the order of side effects may not perform correctly when ported to a new machine or a new compiler.

EXAMPLE

```
i = i++; /* bad - which is done first? = or ++ ? */  
  
++i + i /* bad - is i incremented before second i is accessed? */  
  
s[i++] = t[j++]; /* ok - does not depend upon order */
```

NAME

3.1lexctl - lexical rules for control structure

STANDARD

The standard for formatting is essentially that of Kernighan and Ritchie's The C Programming Language, with a variation in the placement of braces. This layout is the one used in Plauger and Plum's Introduction to Programming in C.

The basic property of this format is shared with all other popular formats for C, namely, the tabbing of subordinate lines:

Each line which is part of the body of a C control structure (if, while, do-while, for, switch) is indented one tab stop from the margin of its controlling line. The same rule applies to function definitions, structure-or-union definitions, and aggregate initializers.

Tabs should be reflected by a uniform amount of white space, preferably four spaces. Four is better than eight because the source listings do not tend to run off the right edge so quickly.

The layout of control structures follows the rule above regarding tabs, and a further rule about braces:

Each opening brace "{" and closing brace "}" appears on a line of its own, tabbed to the same indent as the controlling keyword. The code inside the block is thus indented one tab further than the braces. The explicit exception to this rule is an aggregate initializer which fits entirely on one line.

These rules are illustrated by the following examples:

```
if (a == 1)
    x = y;

if (a == b)
{
    remark("bad value for b", "");
    ++nerrs;
}
else
    subfn(b);

if (!legal(code))
    remark("bad code: ", code);
else if (lookup(code))
    remark("multiple definition: ", code);
else
    install(code, val);
```

```
switch(c)
{
case '\n':
case '\r':
    echo("\n");
    break;
case '\t':
    tabcnt();
    break;
...
default:
    echo(c);
    break;
}

while ((c = getchar()) != EOF)
    putchar(c);

for (p = head; p; p = p->next)
{
    install(p);
    ++syms;
}

FOREVER
    timetest(n);

struct item
{
    struct item *next;
    char *name;
    char *value;
};
```

Nested control structures are formatted by the simple rule that the entire nested structure is indented to the margin of the surrounding body. For example:

```
while ((c = getchar()) != EOF)
{
    if (isspace(c))
        putchar('\n');
    else
        putchar(c);
}
```

Lines within a C source file should fit a listing (or screen) width of 80 characters; any expression that is too long to fit this size should be broken into multiple lines.

A null statement appearing as the body of a loop deserves a line of its own:

```
while (*s++)  
    ;
```

In the test expression of a while, for, do-while, or if, the comparison should be written explicitly, rather than relying upon the default comparison to zero. However, comparison of characters to `'\0'`, booleans to zero or non-zero, and comparison of pointers to NULL can be written as an implicit comparison.

```
while (fgets(stdin, BUFSIZ, buf))  
    process(buf);  
  
while (*s++)  
    ;  
  
if (system(cmd) != 0)  
    fprintf(stderr, "cmd failed\n");
```

Mistaking the single equal-sign assignment operator for the double equal-sign comparison operator is one of the most common C bugs. An embedded assignment in a test expression should always be tested explicitly:

```
while ((*s++ = *t++) != '\0')  
    ;
```

JUSTIFICATION

The choice of a project-wide layout standard pays off in readability of the code and in the ability to create automated text-handling tools like editor scripts, search commands, and "beautifiers".

The set of specific rules presented here was chosen to correct the most common objection to the Kernighan and Ritchie layout (the placement of opening brace) while preserving compatibility with common UNIX layout of function declarations. Surveys by Plum Hall Inc of programmers working in UNIX environments have shown that this style is slightly more popular in practice than that of Kernighan and Ritchie.

ALTERNATIVES

The most common alternatives to this placement of braces are the Whitesmiths style and the "Kernighan and Ritchie" style:

```
while (expr)  
{  
    stmts;  
}  
-- patterned after Whitesmiths (1981)
```

```
while (expr) {  
    stmts;  
}
```

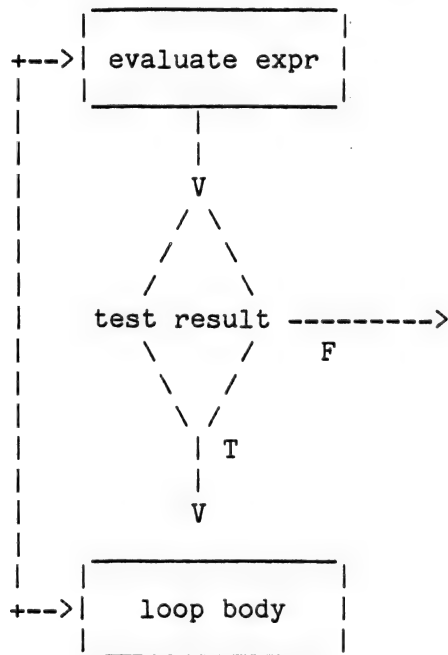
-- patterned after Kernighan and Ritchie (1978)

NAME

3.2while - while and the $N+1/4$ - time loop

STANDARD

Side-effects are explicitly allowed in the test of a while or for. The underlying flow chart shows a box that is executed $N+1$ times whenever the lower box is executed N times, so the loop is known as an $N+1/2$ - time loop. In C, the upper box is only an expression, whereas the lower box is an entire statement, so the loop can be called an " $N+1/4$ - time" loop.



JUSTIFICATION

Such side-effects are one of the strengths of C; they smoothly handle a large proportion of common loops. They almost eliminate the awkward "duplicated read" loops, yet they require no complicated "lookahead" mechanism.

NAME

3.3loopinvar - designing with loop invariants

GUIDELINE

The "invariant condition" of a loop is a "typical picture" of the loop variables, a relationship which

- o is always true at each iteration of the loop, and still true at loop termination; and
- o guarantees that the goal of the loop is attained when the loop terminates.

Consider this familiar loop:

```
while ((c = getchar()) != EOF)
    putchar(c);
```

The invariant condition is "A series of n characters have been read and already printed out in sequence, and one more character c has been read but not yet printed." This condition is true at the start of the loop; and if it is true for n iterations, after we go around the loop once more, it becomes true for $n+1$. The condition guarantees that the goal of the loop is attained at termination, because when c is EOF, all the characters from beginning to end of file have been read and printed.

Graphically, this invariant condition, or "typical picture" looks like this:

Already read: - - ... - c

Already printed: - - ... -

There may be portions of the loop body during which this "typical picture" is not true (as the variables are altered). For example, in the getch loop above, the picture is not strictly true just after c is printed and just before getch delivers the new c . These portions are the "domain of exceptions".

The program will be more readable if the domain of exceptions is minimized.

JUSTIFICATION

Altering the relationship between variables during a loop increases the complexity of the reader's task. The more complex the loop, the greater the need for this uniformity.

"Minimizing the domain of exceptions" gives an explicit justification for the experienced programmer's intuition that certain forms of for loops are preferable; they confine the

domain of exceptions to the for line.

EXAMPLE

```
for (p = head; p; p = p->next)
{
    /* invariant - p points to next item to process */
    ... process p
}

for (pp = &head; *pp && (*pp)->next; pp = &(*pp)->next)
{
    /* invariant - *pp is next item to process, and */
    /* (*pp)->next is the item after it; both exist */
    ... process *pp and (*pp)->next
}
```

NAME

3.4elseif - multiple-choice constructs

GUIDELINE

The "else-if" should be used for multiple-choice constructs whenever the conditions are not mutually exclusive, whenever their order of evaluation is important, or whenever they test different variables. Otherwise, use the "switch".

Do not make up artificial variables just to make use of a "switch".

In either case, each alternative action is tabbed to the same indent.

JUSTIFICATION

Although the "else-if" could always be used instead of "switch", the distinction is helpful for readability. The efficiency of "switch" is sometimes significant. The indentation shows the logical nature of the multiple-choice.

EXAMPLE

```
if (!legal(code))
    remark("bad code: ", code);
else if (lookup(code))
    remark("multiple definition: ", code);
else
    install(code, val);

switch (sc)
{
case EXTERN:
    remark("redefined", "");
    break;
case STATIC:
    printf("...");
    break;
case INTERN:
    printf("...");
    sc = STATIC;
    break;
default:
    remark("unknown sc", "");
    break;
}
```

NAME

3.5control - restrictions on control structures

STANDARD

The goto statement (and hence labels as well) should not be used.

The while loop should be used instead of the do-while loop, except where the logic of the problem explicitly requires doing the body at least once regardless of the loop condition.

If a single if-else can replace a continue, an if-else should be used.

JUSTIFICATION

The goto statement is prohibited for the empirical reason that its use is highly correlated with errors and hard-to-read code, and for the abstract reason that algorithms should be expressed in structures that facilitate checking the program against the structure of the underlying process. (See the Guideline on "structure" for detailed explanation.)

The do-while is discouraged because loops should be coded in such a form as to "do nothing gracefully"; i.e. they should test their looping condition before executing the body.

NAME

3.6structure - matching program structure to problem structure

GUIDELINE

The structure of the data being processed (expressed in augmented "Backus Normal Form" or equivalent notation) should match the structure of the program.

Sequence:

If the data structure is

a b c

(which means "items a, b, and c in that order")

the program structure will be a sequence, such as

process a; process b; process c;

Choice:

If the data structure is

a | b

(which means "choice between a or b")

the program structure will be a conditional, such as

```
if (data is a)
    process a;
else
    process b;
```

Repetition:

If the problem structure is

{ a }

(which means "repetition of the item a")

the program structure will be a loop, such as

```
while (more a)
    process a;
```

OR

```
for (first a; more a; next a)
    process a;
```

JUSTIFICATION

The usefulness of syntax-oriented design has been well-established in practice. Writing down the data syntax helps get the problem clear. The data syntax gives a reliable outline for the first draft of the program. And the process of checking the resulting program against the data syntax aids in the review.

Useful readings are found in Jackson (1975), Warnier (1974), and Wirth (1973).

EXAMPLE

Data structure is
 { a | b | c }

(which means "a repetition of a choice among a, b, and c")

Program structure is
 while (get an item)
 {
 if (its type is A)
 process type-A code;
 else if (its type is B)
 process type-B code;
 else if (its type is C)
 process type-C code;
 else
 error("illegal code: ", code);
 }

NAME

4.1lexfns - lexical rules for functions

STANDARD

External declarations (function definitions and external variables) begin at the left margin (with optional storage class and mandatory defined-type). An explanatory comment is required before each function, and is also at the left margin. A typical one-function file looks like this:

```
#include <stdio.h>
#include <stdlib.h>
TYPEX varx = NNN; /* comment describing varx */

/*
 * comment describing func
 */
TYPE func(a1, a2, a3)
TYPE1 a1, a3;
TYPE2 a2;
{
    <local declarations>

    <statements>
}
```

The blank line is omitted if no local declarations are present.

JUSTIFICATION

This layout is chosen for consistency with the layout of control structures described in 3.1lexctl. The essential feature here, as there, is the uniform indentation of the body and the alignment of the braces above and below the body.

ALTERNATIVES

Some projects choose to put the function return type on the line above the function header:

```
/* comment describing func
 */
TYPE
func(a1, a2, a3)
```

The Whitesmiths (1981) alternative looks like this:

```
/* comment describing func
*/
TYPE func(a1, a2, a3)
    TYPE1 a1, a3;
    TYPE2 a2;
    {
        <local declarations>

        <statements>
    }
```

NAME

4.2headers - project-wide standard header files

STANDARD

All programs within a project must use a standard set of header files.

There will be one designated header-file directory for each project. Each header-file can then be accessed in the project header-file directory.

All definitions of compile-time values and of structure formats should be put into header files if they are shared by more than one file.

Header-files unique to a project or organization are allowed, and are included via

```
#include "file.h"
```

All other header-files will be included via the "angle-brackets" notation:

```
#include <file.h>
```

Name formats specific to one operating system (such as "[1,1]file.h" or <subdir/file.h>) prevent portability to other systems, and are disallowed in portable code.

All #defined names should be in upper-case letters. Structures or unions defined in header files should be given a #defined name.

JUSTIFICATION

During development, it often becomes necessary to rearrange the directory structure. When absolute file names are allowed in #includes, all the source has to be re-edited with the tedious changes.

Different versions of a project can have different header environments without introducing problems of retroactively modifying existing code.

EXAMPLE

Assume that version 1.1 of project a has collected its standard header files into directory /xx/projecta.1.1, which contains stdio.h, stdtyp.h, defs.h, specific.h. In the file pgm.c we will find these #include statements:

```
#include <stdio.h>
#include <stdtyp.h>
#include <defs.h>
#include <specific.h>
```

It will be useful to create a project-wide compile command which specifies this directory as the source of "angle-bracket" header names; assuming that this command is named "proja.cc", we can compile `pgm.c` like this:

```
proja.cc pgm.c
```

To give a #defined name to a structure or union:

```
#define TASK struct task
TASK
{
    TASK *next;
    char *desc;
    long plan, start, finish;
};
```

NAME

4.3files - size for source files

STANDARD

Source files should be no larger than 500 lines.

JUSTIFICATION

Experience has shown that larger files are cumbersome to edit correctly and to maintain.

NAME

4.4includes - put includes at head of file

STANDARD

Each source file will start with its list of #includes grouped at the head of the file, before any declarations or function definitions. After the #includes, next come any #defines needed for the file.

JUSTIFICATION

Header-files form the context for the code in the file. Grouping them together allows quick verification of the list, and facilitates any necessary dependency-checking.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <defs.h>
#include <specific.h>

#define GOODCODE 1
#define BADCODE 0
/* now comes the rest of the program */
```

NAME

4.5stdflags - standard compile-time flags

STANDARD

Standard compile-time flags will be chosen project-wide for environmental issues such as target machine, target operating system, and debug options. Such flags will not be hard-coded into the program (unless required by its environment-specific function), because they can be turned on by command line option.

If the compiler itself provides a built-in flag for target machine, then its set of supported names should be used as a standard.

JUSTIFICATION

Judiciously chosen conditional definitions allow efficient targeting of such issues as word size, byte size, and specific forms for system interfaces such as file control blocks, etc. Selective trace and assertion functions can be included with ease if a flag such as NDEBUB is available to turn off code production when not wanted.

The following list illustrates the possibilities:

UNIX	UNIX target environment
RSX11M	RSX-11/M target environment
CPM	CP/M target environment
ALONE	stand-alone target environment
PDP11	PDP-11 target machine
I80	8080/Z80 target machine
M68000	68000 target machine
VAX	VAX target machine
NDEBUB	turn off debugging options

EXAMPLE

```
$ cc -DNDEBUB -DRSX11M  
  
$ cc -DUNIX
```

NAME

4.6nonest - no nested header files

STANDARD

As a general practice, header-files should not unconditionally #include other header files.

To avoid multiple inclusions of the same header-file, each header-file should begin with a #ifndef that tests whether some #defined symbol has already been defined, and should end with #endif.

JUSTIFICATION

Experience with nested header-files has shown that sequential dependencies lead to interminable conflicts during large-team integration.

Each header file should be included only once during each compilation, and if the includes are nested unconditionally, this property becomes hard to control.

EXAMPLE

specific.h:

```
#ifndef NEEDED-TYPE
... definitions, including NEEDED-TYPE
#endif
```

NAME

4.7noinit - no initializations in header files

STANDARD

Header-files should not contain initialization of anything.

JUSTIFICATION

Putting the initialization in a header-file doesn't make clear which function "owns" the data; i.e., it doesn't localize the "defining instance." Also, multiple source files each including a file containing initializations cannot be compiled by some compilers.

EXAMPLE

defs.h:

```
#define TOKEN short
#define TOKITEM struct tokitem
TOKITEM
{
    char *name;
    TOKEN val;
};
```

Method 1 - If there is one file that contains functions which will always be needed if the data is referenced, put the definition into that file:

tablemgr.c:

```
TOKITEM tokt =
{
    "pin", 1,
    "wire", 2,
    ...
};
...
```

followed by functions which process the data.

Method 2 - If no one file "owns" the data, put the definition in a file of its own, which gives a "data-only" object-file which will be linked when needed by any of several functions:

tokt.c:

```
TOKITEM tokt =
{
    ...
};
```

NAME

4.8coupling - methods of coupling modules together

STANDARD

Data should be local (internal static or external static)
unless global linkage is specifically necessary.

JUSTIFICATION

Integration problems are made more probable by each global
linkage. See Yourdon & Constantine (1978) for lengthier (!)
justification.

NAME

4.9cohesion - cohesion and meaningful functions

STANDARD

Most functions should evidence "functional cohesiveness" (Yourdon & Constantine, 1978), which can be adequately summarized by the following test:

Can the purpose of the function be accurately summarized by a sentence in the form

"specific verb + specific object(s)"?

This sentence should appear in a comment just before the function.

JUSTIFICATION

The logic of the calling module can be verified only if the action of each called function can be grasped without having to resort to line-by-line reading.

EXAMPLE

```
/* error - print unbuffered fatal error message */
void error(s1, s2)
char *s1, *s2;
{
    write(2, s1, strlen(s1));
    write(2, s2, strlen(s2));
    exit(FAIL);
}

/* remark - print unbuffered non-fatal error message */
void remark(s1, s2)
char *s1, *s2;
{
    write(2, s1, strlen(s1));
    write(2, s2, strlen(s2));
}
```

NAME

4.10libfns - file structure for library functions

STANDARD

General-purpose functions (i.e., callable by more than one main) should be placed in separate source files. Each such file should contain at its end a main function which will try out the called function. This test driver should be surrounded by a #ifdef and #endif keyed to a standard compilation flag. This means that functions that are called from more than one file must be documented and maintained as library functions, including a manual-page specification.

To avoid confusion in the calling program, functions should not be given names that conflict with names of functions in the standard library or in project-wide standard libraries. For example: don't make up a new function called "getc" unless it is specifically intended to replace the existing "getc" function.

JUSTIFICATION

Object files are created on a per-source-file basis. Calling programs should not inadvertently link multiple functions when they need only one function. Standardizing the format of tryout functions and putting them into the source reduces the number of files to be maintained and places examples of use in a place where they are most likely to be seen.

EXAMPLE

```
/* scmpr - compare strings (greater, equal, less) */
int scmpr(s, t)
char *s, *t;
{
    while (s && *s == *t)
    {
        s++;
        t++;
    }
    return ((unsigned)*s - (unsigned)*t);
}
#ifdef TRYOUT
main()
{
    if (scmpr("abc", "abc") != 0 || scmpr("ab", "xy") >= 0)
        fprintf(stderr, "no good\n");
}
#endif
```

NAME

4.11portlib - use of portable library

STANDARD

The program should rely on no more about its environment than is given by the project's chosen portable library specification.

Don't assume system-specific I/O formats.

Don't assume availability of system commands.

Don't assume system-specific command-invocations (e.g. UNIX exec).

The files `stdin`, `stdout`, and `stderr` are portable to most libraries and systems, and should be used whenever appropriate:

`stdin`: standard input (should be meaningful for file or terminal)

`stdout`: standard output (either file or terminal)

`stderr`: standard error output (all error messages should be sent to `stderr`).

One problem with the UNIX library is that it is not partitioned into portable and non-portable subsets. A positive way to state this standard is to give a list of the portable functions from the UNIX library. The list given here consists of the functions supported by PHIL, the Plum Hall Interface Library, which supports UNIX functions on all the operating systems supported by Whitesmiths, Ltd. Just because a function does not appear on this list is not a guarantee that it is UNIX-specific; for example, most of the math functions could also be ported. The list can serve, however, as a starting point for portability determinations.

Functions supported by PHIL:

System calls: `sbrk`, `close`, `creat`, `exit`, `lseek` (with some system dependencies regarding text or binary files), `open` (but some systems require extra information about record size), `read`, `unlink`, `write`.

Functions: `ctype` (`isalpha`, `isupper`, `islower`, `isdigit`, `isalnum`, `isspace`, `ispunct`, `isprint`, `isctrl`, `isascii`), `exp`, `log`, `sqrt`, `fclose`, `fflush`, `feof`, `fileno`, `fopen`, `getc`, `getchar`, `fgetc`, `fgets`, `malloc`, `free`, `printf`, `fprintf`, `sprintf`, `putc`, `putchar`, `fputc`, `fputs`, `qsort`, `scanf`, `fscanf`, `sscanf`, `sin`, `cos`, `atan`, `stdio` (but with varying internal details), `string` (`strcat`, `strcmp`, `strncmp`, `strcpy`, `strlen`, `index`), `ungetc`.

Direct file access (lseek, fseek, fopen with "a") is prone to system-dependent problems; portability needs to be determined on a case-by-case basis.

JUSTIFICATION

Occasional use "for convenience" of handy system-specific functions may create serious portability problems later. The new system may not have the same formats or commands as the old system.

NAME

4.12environ - non-portable environment features

STANDARD

In application code, segregate environment-specific code into separate functions. Keep such functions as small and limited as possible.

Another (unfortunate) portability requirement is that external names must be unique in their first 6 characters.

JUSTIFICATION

This minimizes the work in porting to a new environment.

EXAMPLE

```
prefix = "/proj/dictionary/"; /* wrong - wires-in specific system */  
getnam(prefix);             /* ok - dependency is in small function */
```

NAME

4.13fnsiz - suggested size of functions

GUIDELINE

Programs should be designed so that most of the functions will be smaller than 50 lines of source listing.

This guideline does not require artificial splitting of cohesive code into small pieces.

On the other hand, code which consists of nothing but large-function files usually reveals insufficient attention to design prior to launching into code.

JUSTIFICATION

Algorithms are easier to create and to understand if they are built of pieces small enough to be grasped as one concept. Refer to Myers (1978) or Constantine and Yourdon (1978) for further information on design.

The 50-line guideline is derived from empirical observation of well-crafted C code.

NAME

4.14macros - writing macros

STANDARD

Macros should be given names all in upper case. Examples from <stdtyp.h>:

```
MAX(x, y)      MIN(x, y)      ABS(x)
```

When writing macros, be sure to put parentheses around each of the parameters in the replacement text, and around the entire replacement text, to guard against precedence surprises:

```
#define SQUARE(x)  x * x      /* wrong */
#define SQUARE(x)  ((x) * (x)) /* right */

f = z / SQUARE(y + 1);
```

When documenting macros with a manual page, be sure to indicate a "BUGS" or "CAUTION" entry, warning that the name is a macro and that no side-effects should appear on its arguments.

JUSTIFICATION

These problems with precedence and side-effects cause macros to behave differently from functions. The user must be protected when possible, and otherwise warned.

ALTERNATIVES

Prevalent practice has been to give macros lower-case names if they are maintained as part of a library (with manual pages and function-like invocations). For example, <stdio.h> defines `getchar`, `putchar`, `feof`, etc., in lower case. The average C programmer occasionally forgets that these are macros and attaches side-effects to their arguments. Worse, some compilers (e.g. early Whitesmiths, 2.1 and prior) will expand a macro name invocation even if it is not followed by "(". Thus, if `min` is defined as a macro, and the programmer happens to name some variable `min`, each occurrence of the name will be treated as a macro call with null arguments, invariably producing bizarre and confusing results.

NAME

5.1compilers - avoiding non-portable compiler features

STANDARD

Programs should avoid compiler-dependent features.

(See Chapter 6 for lists of known compiler-dependent features.)

JUSTIFICATION

The choice of compilers used for a product will be based on global issues such as license arrangements, reliability of generated code, and variety of target machines. Such choice can be severely constrained by proliferation of compiler-dependent code.

ALTERNATIVES

Some projects may see a need for compiler-dependent features. In these cases, a task force should plan for the possible migration to other compilers, and devise a method for porting the programs to these other compilers.

NAME

5.2comments - suggested use of comments

GUIDELINE

A function which performs a simple transformation on its arguments may adequately be documented by a one-line "verb + objects" header comment. For larger or more complicated functions, a good explanatory comment should describe the important data structures used and should depict the control flow with a few lines of "pseudo-code" or "program design language".

Ideally, the use of meaningful variable names and clear control structures will eliminate the need for line-by-line commenting, but each "paragraph" or logical grouping of statements will be made more readable by a comment prior to the block, indented at the same level as the statements.

Where an individual statement needs clarification (even after attempts to clarify the code itself), its explanatory comment is placed at the right of the line being explained, separated by one tab stop from the end of the line. This form of comment is also useful for describing arguments on their declaration statement.

A related guideline for other variable declarations is that a declaration which specifies an initial value will declare only one variable, and a comment can conveniently be placed at the right of the line.

Modifications made after the initial release of a product will be documented by a similar comment on the right of the line, formatted in a standard way. A related recommended practice is for each subproject to keep a numbered log of change-requirements, and to refer to the appropriate number in the comment.

JUSTIFICATION

Following the style of examples shown in programming textbooks usually gives insufficient clues about what the function does; this is because textbooks actually contain the "comments" in the text of the book.

One good practical test of the amount of commenting needed is "can the reviewer understand the function without detailed coaching from the author".

It must be recognized that there is such a thing as over-commenting, which restates the obvious and slows down comprehension by sheer weight of reading matter. Therefore,

individual judgement is needed.

EXAMPLE

One function from a larger file:

```
static ITEM *phead = NULL; /* head of singly-linked list */
/*
 * cbsave - save a composite-bill item on list
 *
 * look for the identifier s on the list
 * if found
 *     increment its quantity field
 * else
 *     allocate a new item
 *     splice it into the list
 */
void cbsave(s, n)
char *s;          /* identifier field of a cb item */
int n;            /* quantity field of a cb item */
{
    ITEM *p, *q;   /* pointers to current item */
    ITEM **from;   /* address of the link that points to p */

    for (p = phead, from = &phead;
         p && strcmp(s, p->part_no) > 0;
         from = &p->next, p = p->next)
        ;
    if (p && strcmp(s, p->part_no) == 0) /* Release 1.1 mod #7 */
        p->qty += n;
    else
    {
        q = malloc(sizeof ITEM);
        q->next = p, p = q;
        p->qty = n;
        p->part_no[0] = '\0';
        strncat(p->part_no, s, sizeof p->part_no);
        *from = p;
    }
}
```

NAME

5.3specs - specifications

STANDARD

A program must be unmistakably matched to an external specification (such as a library manual page, like this page).

The program must meet all the demands of its specification (summarizable as "the program must work").

If a program consists of multiple source files, there must be a visible way to tell what files these are (since its specification applies to the entire set). On UNIX, one standard way to show the correspondence is in the "Makefile" which shows the command which compiles the main function. For example, the command

```
c mainpgm.c file2.o file3.o
```

shows the three files involved.

JUSTIFICATION

If a program isn't required to work correctly, then we can make it as small, as fast, or as readable as we want.

That it works correctly is the fundamental criterion.

Verifying its correctness requires a specification to check it against.

NAME

5.4reviews - code reviews

STANDARD

If a program is to be part of a released product, the program must be reviewed by one or more people other than its author.

The names of author and reviewer(s) must be documented in a comment.

The reviewer's concurrence means "I have read all the code and its corresponding specification. As best I can tell, the program is understandable, meets its specification, and conforms to all applicable standards. I would be able to maintain it."

Requisite efficiency may be considered a part of the specification, and should be spelled out explicitly in critical cases.

The review process applies to the entire source file, rather than to each function separately; the source file is the basic entity for software administration.

JUSTIFICATION

There is a class of bugs known as "blind spots" which will never be found by the author. This means that all applications that require CORRECTNESS in the product must have some procedure for review by someone other than the author.

It is desirable to have a procedure such as "code review" or "walkthrough" which formalizes the review process, but even without formal processes, the interests of future maintainers demand that the program must be understandable by someone other than its original author.

NAME

5.5defensive - defensive programming

STANDARD

Programs should not "blow up" or behave unpredictably in the face of out-of-bounds data. This does not mean that programs are required to explicitly diagnose every out-of-bounds condition -- only that they should behave sensibly. Nor does it mean that every function is required to perform bounds-checking. Somewhere within the source file must be found the code that will protect against out-of-bounds data.

JUSTIFICATION

Specifications should not be burdened with case-by-case requirements that programs behave sensibly; this should be assumed as a quality of professional software.

On the other hand, requiring bounds-checking on every statement or even every function can be grossly inefficient. The choice of the source file as the locus of responsibility is predicated on the choice made in 5.4 reviews that the source file is the basic entity for software review and administration.

NAME

6.1v7 - Non-standard features of UNIX V7 and III compilers

NON-STANDARD FEATURES

Struct assignment, struct arguments, and struct returned values are allowed.

#include "filename" searches

- (1) source-file directory,
- (2) optional command-line directory-list,
- (3) standard places.

#include <filename> searches

- (1) optional command-line directory-list,
- (2) standard places.

Structures containing bit fields can't be initialized.

Some versions accept unsigned char data type, and some accept unsigned short.

The enum data type is defined.

In UNIX System III, the void type is allowed for functions that return no value.

The modern treatment of struct-or-union member names is as follows: Each struct-or-union has its own table of member names. The operators . and -> require an appropriate struct-or-member type on the left, and a member name from that struct-or-member on the right. Most V7 (and later) compilers seem to follow this modern treatment (the same as Whitesmiths "-m" flag) even though V7 manuals do not mention it.

The permissive linker handling of external data names initialized more than once, or not at all, is probably not guaranteed for all UNIX-derived systems. The conservative course is to ensure that each external data name is initialized exactly once.

NAME

6.2ws - Non-standard features of Whitesmiths compiler

NON-STANDARD FEATURES

Static variables MUST have explicit initializers outside functions.

External variables must be initialized in exactly ONE object file.

Backslash at end-of-line splices the next line to this one.

A union may be initialized.

All struct tags are in a name space separate from all union tags.

A preprocessor macro invocation, e.g. swap(a, b), must be written all on one line, possibly spliced with backslash.

#include "filename" only searches current directory.

#include <filename> only searches one standard directory, in Versions 2.0 and prior. In Version 2.1 (and later), #include <filename> searches whatever list of directories was specified in the arguments to the preprocessor.

The sizeof operator is disallowed in #if expressions.

The identifier "_" is reserved.

Leading whitespace is allowed before preprocessor "#".

Right-shifting a signed quantity gives a guaranteed sign-propagation.

Version 2.0 (and later) defines some non-standard types: unsigned char, unsigned short, unsigned long. Using these types enhances machine-portability at the expense of compiler-portability.

Type-widening is applied in several contexts:

1. In formal parameter declarations, the type of variables is widened up to a preferred size. For example, a parameter that is declared char is actually understood as an int variable. (Versions 2.1 and prior.)
2. Casts will not truncate smaller than a preferred size. Thus, (char)x behaves the same as (int)x. (Versions 2.1 and prior.)

3. Function returned values will not be truncated smaller than a preferred size. Thus, the declaration

```
char f()
```

is rewritten by the compiler into

```
int f()
```

(Versions 2.1 and prior.)

4. Register declarations are widened to a preferred size. Thus the declaration

```
register char x;
```

is understood by the compiler as

```
register int x;
```

(Versions 2.1 and prior.)

A macro name cannot be used as a variable name; the compiler replaces the name even if it is not followed by "(". (Versions 2.1 and prior.)

The modern treatment of struct-or-union member names is as follows: Each struct-or-union has its own table of member names. The operators . and -> require an appropriate struct-or-member type on the left, and a member name from that struct-or-member on the right. This modern treatment is obtained with the "-m" flag. Omitting this flag gives the older treatment described in Kernighan and Ritchie. Since most UNIX compilers seem to follow the modern treatment (see 6.1v7), most projects should add the "-m" flag to their compilers. (Whitesmiths own source code may need compilation without the "-m" flag.)

NAME

6.3v6 - Non-standard features of UNIX V6 compiler

NON-STANDARD FEATURES

The following are not implemented:

bit fields, short integers, unsigned integers, long integers, casts, unions, #if, #line, assignment operators in the form "op=", static external declarations (local to a file), register arguments.

Initializers are not allowed to have the "="; i.e., this is illegal:

```
int x = 1;
```

It is legal to give this declaration in either of these forms:

```
int x 1;  
int x {1};
```

"#" must be the first character of source file if preprocessor is wanted.

A structure is initialized as an array of ints.

NAME

6.4refs - references

REFERENCES

1. Jackson, M. A. Principles of Program Design. Academic Press, 1975.
2. Kernighan, Brian W., and Dennis M. Ritchie. The C Programming Language. Prentice-Hall, 1978.
3. Kernighan, Brian W., and P. J. Plauger. The Elements of Programming Style. McGraw-Hill, 1978.
4. Myers, Glenford J. Composite/Structured Design. Van Nostrand Reinhold, 1978.
5. Warnier, Jean Dominique. Logical Construction of Programs. Van Nostrand Reinhold, 1974.
6. Wirth, Nicklaus. Systematic Programming. Academic Press, 1973.
7. Yourdon, Edward, and Larry L. Constantine. Structured Design. Yourdon Press, 1978.
8. Whitesmiths, Ltd. C Compiler Manual. Whitesmiths, 1981.
9. Jensen, Kathleen, and Nicklaus Wirth. PASCAL User Manual and Report. Springer-Verlag, 1974.
10. cummings, e. e. 100 selected poems. Grove Press, 1959.
11. Plum, Thomas. C Programming Standards and Guidelines: Version W (Whitesmiths). Plum Hall Inc, 1982.

NAME

6.5index - index

6.3
#define 1.3, 1.4, 4.2, 4.4
#if 6.3
#ifdef 4.10
#ifndef 4.6
#include 4.2, 4.4, 4.6, 6.1, 6.2
#line 6.3
&& 2.2
-m option 6.2
50 lines, per function 4.13
500 lines, per file 4.3
80 characters, per line 3.1
<file.h> 4.2, 6.1, 6.2
<std.h> 1.3
?: 2.2
abbreviations 1.2
address-of operator 1.6
addresses, passed as arguments 1.6
aggregate initializer 3.1
alignment 1.9
alloc() 1.9
ambiguity, of -= 2.1
angle-brackets 4.2, 6.1, 6.2
arguments 2.1
arrays of structures 1.9
assignment 2.1, 2.5, 3.1
assignment, op= 6.3
backslash 6.2
Backus Normal Form 3.6
binary data 1.6
binary operators 2.1
bit fields 6.1, 6.3
bitwise operators 1.3, 1.5, 2.3
blank line 4.1
blind spots 5.4
BNF 3.6
braces 3.1, 4.1
byte length 1.5
byte ordering 1.6
canonical ordering 1.6
cast 1.5, 6.3
change-requirements 5.2
char 1.5
char, unsigned 6.2
character constants 1.7
code review 5.4
cohesion 4.9
comma 2.1, 2.2
comments 1.1, 4.1, 4.9, 5.2

- comparison to zero 3.1
- compile command, project-wide 4.2
- compile-time flags, standard 4.4
- compile-time values 4.2
- compiler features, non-portable 5.1
- conditional operator 2.1, 2.2
- constants, character 1.7
 - maintainability of 1.4
 - multi-character 1.7
- continue 3.5
- control structure 3.1, 3.5
- control structure, lexical rules, for 3.1
- conversions, pointer 1.9
- coupling 4.8
- data size 1.5
- data structure 3.6
- data-only object-file 4.7
- debug options 4.4
- declarations 1.1
- decrement 2.5
- defined-types 1.3
- defining instance 4.7
- directory structure 4.2
- do nothing gracefully 3.5
- do-while 3.1, 3.5
- domain of exceptions 1.2, 3.3
- efficiency 5.4
- else-if 3.4
- environment 4.11
- error 4.9
- evaluation order 2.2
- exceptions 0.1
- exceptions, project-wide 0.1
- external variables 1.1, 4.1, 4.12
- external variables, and initializers 6.2
- FAIL 1.3
- file names 4.2
- file structure, for library functions 4.10
- floating data 1.6
- for 2.1, 3.1
- function definition 3.1
- function names, and parentheses 2.1
- functions, general-purpose 4.10
 - lexical rules, for 4.1
 - return type 4.1
 - size of 4.13
- general-purpose functions 4.10
- GLOBAL 4.8
- goto 3.5
- guidelines 0.1
- header files, and initialization 4.7
 - nested 4.6
 - project-wide standard 4.2
- header-file 4.4
- header-files 1.4

- if 2.1, 3.1
- if-else 3.5
- increment 2.5
- indentation 3.1, 3.4, 4.1, 5.2
- initialization 5.2
- initializations, in header files 4.7
- initializers 1.1
- initializers, no = 6.3
- int 1.8
- invariants 1.2, 3.3
- Jackson 3.6
- Jensen and Wirth 3.1, 4.1
- Kernighan and Ritchie 3.1, 4.1, 4.9
- keywords 2.1
- label 3.5
- lexical rules 1.1
- lexical rules, for control structure 3.1
 - for functions 4.1
 - for operators 2.1
 - for variables 1.1
- library functions, file structure for 4.10
- library, portable 4.11
- lint 1.8
- LOCAL 4.8
- long 1.5, 6.3
- long, unsigned 6.2
- loop invariants 3.3
- lower case 1.1, 4.14
- macros 4.14, 6.2
- magic numbers 1.4
- Makefile 5.3
- mandatory 0.1
- manifest constants 1.4
- manual page 4.10, 4.14, 5.3
- meaningful functions 4.9
- mixed case 1.1
- multi-character constants 1.7
- multiple-choice 3.4
- Myers 4.13
- names, meaningful 1.2
 - redefined 1.2
- nested control structure 3.1
- non-portable environment 4.12
- NULL 1.9
- null statement 3.1
- operators 2.1
- order of evaluation 2.2
- order of side effects 2.5
- paragraph 5.2
- parentheses 2.1, 4.14
- parentheses, and bitwise operators 2.3
- pointer 1.6, 1.8
- pointer conversions 1.9
- portability 1.3, 1.5, 1.6, 2.2, 4.2, 5.1, 6.2
- portable library 4.11

precedence 2.1, 2.3, 4.14
primary operators 2.1
problem structure 3.6
program design language 5.2
program structure 3.6
project-wide exceptions 0.1
pseudo-code 5.2
pseudo-divide 2.4
register, argument 6.3
remark 4.9
return 2.1
review, code 5.4
right shift 2.4, 6.2
rvalues, size of 1.5
scmpr() 4.10
semicolon 2.1
sequence of declarations 1.1
short 1.5, 6.2, 6.3
short names, standard 1.2
side effects 3.2, 4.14
side effects, order of 2.5
sign-extension 2.4
size, of functions 4.13
sizeof 1.9, 2.1, 6.2
source files 4.10, 4.4, 5.3, 5.4
source files, size for 4.3
spaces 2.1
specifications 4.10, 5.3
standard defined-types 1.3
standard short names 1.2
standards and guidelines 0.1
standards, exceptions 0.1
static, and initializers 6.2
 external 6.3
structure 3.1
structure formats 4.2
structure, arguments 6.1
 array of 1.9
 assignment 6.1
 initialization 6.3
 returned value 6.1
SUCCEED 1.3
switch 2.1, 3.1, 3.4
syntax-oriented design 3.6
target machine 4.4
target operating system 4.4
truncation 1.5
TRYOUT 4.10
two's-complement 2.4
typecast 1.5, 6.3
typical picture 3.3
unary operators 2.1
union 3.1, 6.2, 6.3
UNIX, V6 1.1, 6.3
 V7 6.1

unsigned 6.2, 6.3
unsigned, and right shift 2.4
upper-case 1.4, 4.14, 4.2
variable names 1.1, 1.2
verb and objects 4.9
Warnier 3.6
while 2.1, 3.1, 3.2, 3.5
Whitesmiths 3.1, 6.2
width, of listing 3.1
Wirth 3.6
word size 1.3, 1.5
Yourdon and Constantine 4.13, 4.8, 4.9
zero, comparison to 3.1
|| 2.2

